

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C08

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monade - privire de ansamblu

Despre intenție și acțiune

[1] S. Peyton-Jones, *Tackling the Awkward Squad*: ...

- [1] A purely functional program implements a function; it has no side effect.
- [1] Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, ...

Exemplu

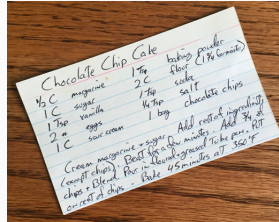
```
putChar :: Char -> IO ()  
Prelude> putChar '!'
```

reprezintă o comandă care, **dacă va fi executată**, va afișa un semn de exclamare.

Mind-Body Problem - Rețetă vs Prăjitură



c :: Cake



r :: Recipe Cake

IO este o rețetă care produce o valoare de tip **a**.

Motorul care citește și execută instrucțiunile **IO** se numește **Haskell Runtime System** (RTS). Acest sistem reprezintă legătura dintre programul scris și mediul în care va fi executat, împreună cu toate efectele și particularitățile acestuia.

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.
- "All told, a monad in X is just a monoid in the category of endofunctors in X, with product \times replaced by composition of endofunctors and unit set by the identity endofunctor."

Saunders Mac Lane, Categories for the Working Mathematician, 1998.

- O monadă este un burrito. <https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/>



<https://twitter.com/monadburritos>

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

știind x , obținem **direct** y

- Funcție îmbogățită: $x \mapsto$



știind x , putem să **extragem** y și să producem un **efect**

Referințe:

<https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/>

<https://bartoszmilewski.com/2016/11/30/monads-and-effects/>

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu:

Folosind tipul **Maybe a**

```
data Maybe a = Nothing | Just a
```

```
f :: Int -> Maybe Int
```

```
f x = if x < 0 then Nothing else Just (x+1)
```

Cum putem aplica f de două ori peste 0 și să obținem Just 2?

Clasa de tipuri Monad

```
class Applicative m => Monad m where  
  (>=)  :: m a -> (a -> m b) -> m b  
  (>>)  :: m a -> m b -> m b  
  return :: a -> m a
```

În Haskell, monada este o clasă de tipuri!

Clasa **Monad** este o extensie a clasei **Applicative**!

- $m\ a$ este tipul **comenzilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m\ b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- \geq este operația de „secvențiere” a comenzilor

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

Exemplu:

Folosind tipul **Maybe a**

```
data Maybe a = Nothing | Just a
```

```
f :: Int -> Maybe Int
```

```
f x = if x < 0 then Nothing else (Just x+1)
```

Cum putem aplica f de două ori peste 0 și să obținem Just 2?

```
> f 0 >>= f
```

Asociativitate și element neutru

Operația `>>=` este asociativă și are element neutru **return**

- Element neutru (la dreapta):

$$(\text{return } x) \gg= g = g \ x$$

- Element neutru (la stânga):

$$x \gg= \text{return} = x$$

- Asociativitate:

$$(f \gg= g) \gg= h = f \gg= (\backslash x \rightarrow (g \ x \gg= h))$$

Un exemplu care citește și afișează un string

```
getLine >=> putStrLn
```

```
(>=>)  :: m a -> (a -> m b) -> m b
```

```
getLine  :: IO String
```

```
putStrLn :: String -> IO ()
```

De ce nu este suficient `fmap`?

Exemplu: `putStrLn <$> getLine`

```
<$> :: Functor f => (a -> b) -> f a -> f b
```

```
getLine :: IO String
```

```
putStrLn :: String -> IO ()
```

```
-- in exemplul nostru, b devine IO ()
```

```
--                                [1] [2] [3]
```

```
putStrLn <$> getLine :: IO (IO ())
```

[1] **IO** din exterior reprezintă efectul pe care **getLine** trebuie să îl producă pentru a obține un **String** introdus de utilizator

[2] **IO** din interior reprezintă efectul care s-ar produce dacă **putStrLn** a fost evaluat

[3] **()** este tipul unitate întors de **putStrLn**

De ce nu este suficient `fmap`?

```
putStrLn <$> getLine :: IO (IO ())
```

Trebuie să unim efectele lui `getLine` și `putStrLn` într-un singur efect **IO!**

```
import Control.Monad (join)
join :: Monad m => m (m a) -> m a
join $ putStrLn <$> getLine
```

Notăția **do** pentru monade

$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$(\gg) :: m\ a \rightarrow m\ b \rightarrow m\ b$

Notăția cu operatori	Notăția do
$e \gg= \backslash x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e \gg= \backslash _ \rightarrow \text{rest}$	e rest
$e \gg \text{rest}$	e rest

```
binding' :: IO ()
```

```
binding' =
```

```
    getLine >>=
```

```
        putStrLn
```

```
binding :: IO ()
```

```
binding = do
```

```
    name <- getLine
```

```
    putStrLn name
```

Notăția do pentru monade

```
twoBinds' :: IO ()
twoBinds' =
    putStrLn "name_pls:" >>
    getLine >>=

    \name ->
        putStrLn "age_pls:" >>
        getLine >>=

    \age ->
        putStrLn ("y_helo_thar:_ "
            ++ name ++ "_who_is:_ "
            ++ age ++ "_years_old.")
```

Notăția do pentru monade

```
twoBinds :: IO ()  
twoBinds = do  
    putStrLn "name_pls:"  
    name <- getLine  
  
    putStrLn "age_pls:"  
    age <- getLine  
  
    putStrLn ("y_helo_thar:_"  
              ++ name ++ "_who_is:_"  
              ++ age ++ "_years_old.")
```


Notăția **do** pentru monade

$(\gg=) :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$(\gg) :: m\ a \rightarrow m\ b \rightarrow m\ b$

Notăția cu operatori	Notăția do
$e \gg= \backslash x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e \gg= \backslash _ \rightarrow \text{rest}$	e rest
$e \gg \text{rest}$	e rest

De exemplu

$e1 \gg= \backslash x1 \rightarrow e2 \gg e3$

devine

Notăția **do** pentru monade

$(\gg=) \quad :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$(\gg) \quad :: m\ a \rightarrow m\ b \rightarrow m\ b$

Notăția cu operatori	Notăția do
$e \gg= \backslash x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e \gg= \backslash _ \rightarrow \text{rest}$	e rest
$e \gg \text{rest}$	e rest

De exemplu

$e1 \gg= \backslash x1 \rightarrow e2 \gg e3$

devine

do

$x1 \leftarrow e1$
 $e2$
 $e3$

Notăția do pentru monade

$(\gg=) \quad :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$(\gg) \quad :: m\ a \rightarrow m\ b \rightarrow m\ b$

De exemplu

$e1 \gg= \backslash x1 \rightarrow e2 \gg= \backslash x2 \rightarrow e3 \gg= \backslash_ \rightarrow e4 \gg= \backslash x4 \rightarrow e5$

devine

Notăția do pentru monade

$(>>=) \quad :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

$(>>) \quad :: m\ a \rightarrow m\ b \rightarrow m\ b$

De exemplu

$e1 >>= \backslash x1 \rightarrow e2 >>= \backslash x2 \rightarrow e3 >>= \backslash_ \rightarrow e4 >>= \backslash x4 \rightarrow e5$

devine

do

$x1 \leftarrow e1$

$x2 \leftarrow e2$

$e3$

$x4 \leftarrow e4$

$e5$

Functor și Applicative definiți cu return și >=>

```
instance Monad M where
```

```
    return a = ...
```

```
    ma >=> k = ...
```

```
instance Applicative M where
```

```
    pure = return
```

```
    mf <*> ma = do
```

```
        f <- mf
```

```
        a <- ma
```

```
        return (f a)
```

```
    -- mf >=> (\f -> ma >=> (\a -> return (f a)))
```

```
instance Functor M where
```

```
    fmap f ma = pure f <*> ma
```

```
    -- ma >=> \a -> return (f a)
```

```
    -- ma >=> (return . f)
```

Exemple de efecte laterale

I/O	Monada IO
Parțialitate	Monada Maybe
Excepții	Monada Either
Nedeterminism	Monada [] (listă)
Logging	Monada Writer
Stare	Monada State
Memorie read-only	Monada Reader

Monada Maybe (a funcțiilor parțiale)

```
data Maybe a = Nothing | Just a
```

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b  
return :: a -> Maybe a
```

```
instance Monad Maybe where  
    return = Just
```

```
    Just va >>= f = f va  
    Nothing >>= _ = Nothing
```

Monada Maybe – exemplu

```
radical :: Float -> Maybe Float
radical x
    | x >= 0 = return (sqrt x)
    | x < 0  = Nothing

-- a * x^2 + b * x + c = 0
solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return (negate c / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```


Monada Maybe – exemplu

Atenție! Acest exemplu folosește și monada listă care o să fie explicată în cursul următor.

```
-- a * x^2 + b * x + c = 0
solEq2All :: Float -> Float -> Float -> Maybe [Float]
solEq2All 0 0 0 = return [0]
solEq2All 0 0 c = Nothing
solEq2All 0 b c = return [negate c / b]
solEq2All a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    let s1 = (negate b + rDelta) / (2 * a)
    let s2 = (negate b - rDelta) / (2 * a)
    return [s1,s2]
```

Monada Either (a excepțiilor)

```
data Either err a = Left err | Right a
```

```
(>>=) :: Either err a -> (a -> Either err b) ->  
      Either err b
```

```
return :: a -> Either err a
```

```
instance Monad (Either err) where  
  return = Right
```

```
    Right va >>= f = f va
```

```
    err >>= _ = err
```

```
-- Left verr >>= _ = Left verr
```

Monada Either – exemplu

```
radical :: Float -> Either String Float
radical x
  | x >= 0 = return (sqrt x)
  | x < 0  = Left "radical:_argument_negativ"

-- a * x^2 + b * x + c = 0
solEq2 :: Float -> Float -> Float -> Either String Float
solEq2 0 0 0 = return 0
solEq2 0 0 c = Left "ecuatie:_fara_solutie"
solEq2 0 b c = return (negate c / b)
solEq2 a b c = do
  rDelta <- radical (b * b - 4 * a * c)
  return ((negate b + rDelta) / (2 * a))
```

Pe săptămâna viitoare!